Static Code Verification Through Process Models

Sebastiaan Joosten(✉) and Marieke Huisman

University of Twente, Enschede, The Netherlands
sjcjoosten@gmail.com

Abstract. In this extended abstract, we combine two techniques for program verification: one is Hoare-style static verification, and the other is model checking of state transition systems. We relate the two techniques semantically through the use of a ghost variable. Actions that are performed by the program can be logged into this variable, building an event structure as its value. We require the event structure to grow incrementally by construction, giving it behavior suitable for model checking. Invariants specify a correspondence between the event structure and the program state. The combined power of model checking and static code verification with separation logic based reasoning, gives a new and intuitive way to do program verification. We describe our idea in a tool-agnostic way: we do not give implementation details, nor do we assume that the static verification tool to which our idea might apply is implemented in a particular way.

1 Introduction

We recognise two powerful ways of reasoning about concurrent and distributed programs: one can use concurrent separation logic and Hoare-style reasoning, or one might see the program and its environment as a state transition system and use model checking. For reasoning about concurrent and distributed systems, Hoare-style reasoning [7] has been applied successfully [5]. Using different forms of transition systems to model concurrent and distributed systems goes back a long way [9] and can often be a more intuitive method. Neither of these approaches individually is a silver bullet for reasoning about concurrent and distributed programs. Our contribution lies in presenting how to get both techniques: We present a technique to describe program behavior through an event structure, and use properties provable through model checking those descriptions to verify the program using Hoare-style reasoning. Although we do not know whether this combination actually strengthens the verification framework (in the sense of being able to prove more properties), we do believe that the combination makes the verification framework easier to use, by virtue of being able to combine the two techniques as needed.

As a running example, consider this pseudocode that uses a simple spinlock:

```
global boolean la; // true if lock is available, thus not locked
void thread(){
    // acquire lock
    boolean success = false;
    while (!success){
      success = compare_and_swap(la,true,false);
    }
    assert (la == false); // we have the lock
    // release lock
    la = true;
}
```

We will give invariants I that describe that la is set and unset as a lock and unlock action is preformed. In particular, we focus on showing that the Hoare-logic statement $\{I\}$compare_and_swap(l,true,false)$\{I\}$ is valid. We will use model checking to show this.

The approach to verification of programs is as follows: we first tie the behavior of the program to an event structure, by adding ghost code that builds the event structure. For the example, this describes the lock and unlock events. This event structure is then, through an invariant, constrained to a process that describes allowed behaviors of the program. For the example, the process is one where locks and unlocks alternate arbitrarily often. To automatically prove this invariant, we use additional invariants that describe the relation of program variables to the event structure. For the example, this ties the value of the variable la to the state of the event structure. By using techniques from model checking, we can then prove both invariants. This allows us to then use the invariants in a Hoare-logic style proof.

The example is a typical concurrent program: the method of synchronisation, a compare_and_swap, assumes a single shared memory, and there are no send and receive commands as one expects in distributed code. We present concurrent code for simplicity and presentation purposes. The principle to combine reasoning about code with the use of transition systems directly generalizes to distributed systems. Typical challenges one encounters with distributed systems, like heterogeneity, faults in links or nodes, and dynamic topologies, are orthogonal to this paper. Existing solutions for dealing with faults [6] or dynamic topologies [13] use abstract models, describing them in some form of transition system. We therefore consider these challenges and solutions out of scope, but highly relevant: proving the same properties at the code level requires making a link between the abstract level and the actual code, which we demonstrate here.

This paper illustrates an idea on how to verify examples like the one mentioned above, rather than giving an implementation. We hope it is an inspiration to authors of verification tools that apply Hoare-style reasoning. Indeed, we ourselves intend to implement the ideas outlined here in Vercors [3], which is such a tool. However, the best way to implement the idea varies widely from tool to tool. We therefore consider it useful to describe the general idea in a paper separate from its implementation details.

As we are combining Hoare-style proofs and model checking, there is plenty of related work to mention. We describe the work that is most closely related to this paper: concurrent separation logic, model checking, and abstract models.

Concurrent Separation Logic. Hoare-style reasoning is proving a Hoare-triple $\{P\}S\{Q\}$ for the program S. The triple $\{P\}S\{Q\}$ states that if $\{P\}$ holds before running S, then $\{Q\}$ holds after the execution of S. Separation logic gives a default notion of how the program can be composed: The frame-rule states that if $\{P\}S\{Q\}$ is proven, then also $\{R * P\}S\{R * Q\}$ holds. Here $R *$ indicates that the environment in which S is run can be extended by a disjoint set of properties R. In many practical examples, different threads work on different memory, and concurrent separation logic gives a convenient way to reason about such programs.

Concurrent separation logic can sometimes be adapted to new or unconventional synchronisation mechanisms as well. The thesis by Amighi nicely illustrates that some synchronisation mechanisms can fit into a separation-logic based line of reasoning [2]. A clever encoding of the synchronisation primitives allows us to reason about programs that use them. In some cases, one can even verify some of the synchronisation mechanisms themselves. In contrast to Amighi's thesis, this work presents a uniform way to verify those synchronisation mechanisms, as well as those for which verification has not been possible with techniques from concurrent separation logic.

Model Checking. If a program is modeled as a state machine, model checking can be used to establish which properties hold. Not all programs lend themselves to this: unbounded loops, recursion and weak-memory models pose challenges. Recent advances have made model checkers more powerful in these areas: Komuravelli et al. show how to use SMT-based model checkers for the verification of loops and recursion [8]. Model checking has been adapted to reason with weak-memory models effectively [1,15]. Calcagno et al. use model checking in a modular way, modeling the environment of a thread such that it can be used as a specification of that thread later [4]. This work aims to bring these recent improvements of model checkers to the static code verification domain.

Abstract Models. This paper generalizes previous work on abstract models as proposed by Oortwijn et al. [10,11]. In the work of Oortwijn, the contract for a method states which actions may or will be taken by that thread. We generalize this by storing the associated actions in a ghost variable.

An important difference between our work and the work of Oortwijn is how invariants are treated: in the work of Oortwijn et al. pre- and postconditions are specified for actions. From these conditions, some invariants follow. We start by specifying invariants, from which pre- and postconditions follow. In particular, we specify processes in the form of an invariant as well, simplifying their presentation. Simultaneously, we potentially increase the applicability of verification methods.

Contribution. We combine separation logic and model checking by adding a ghost variable that expresses part of an event structure of the program. A ghost variable is a specification-only variable, for the sake of static verification. It can help describe the program state, but it should not exist at runtime. As such, ghost variables aren't allowed to influence the program flow. However, ghost variables can be used to state invariants and properties of the program conveniently.

In contrast to conventional ghost variables, we introduce event structure ghost variables in a way that it gives us additional properties. An event structure is a partially ordered multiset of actions. Our event structure ghost variables can only be updated by adding events at the end of the structure: events that are added must be larger than some maximal element. This restriction means that events are never removed, the structure never shrinks, and for each event, the set of events preceding it is fixed throughout the program execution.

The power of introducing such a variable comes from its use in invariants. An invariant is a property that must be satisfied initially, and is preserved by each atomic action. Consequently, one assumes the invariant is satisfied before an atomic action. For our lock example, we could describe that our event structure must be a prefix of $\mathsf{lock}, \mathsf{unlock}, \mathsf{lock}, \ldots$. A model checker can then tell us that if we are in a state in which lock just happened, the next action will be unlock. Similarly, we can say that la is true if and only if the event structure is in the language $(\mathsf{lock}\ \mathsf{unlock})^*$. The combination of these invariants lets us reason about attainable values of program variables.

The contribution of this work is the description of an event structure ghost variable, as well as an indication on how one might implement them into static checkers and model checkers. By using a ghost variable, as in this work, we naturally tie into existing verification paradigms.

Section 2 describes the event structure variable we introduce. Section 3 describes how such a variable can be related to a process. In Sect. 3.3 we give ways in which to tie the variable in with a system talking about invariants. We conclude in Sect. 4.

2 Using an Event Structure Variable

This section introduces event structures. The purpose of event structures is to capture a program run at an abstraction level that fits reasoning about processes, which we introduce later.

In what follows, we assume that a set of actions $\mathcal{A}$ is given. The purpose of these actions is that they will correspond to program events, but this is left to the modeler: Event structures capture actions as a partially ordered multiset of actions (actions can occur multiple times). The ghost code describes how the actions are added to the event structure. We proceed by defining what an event structure is.

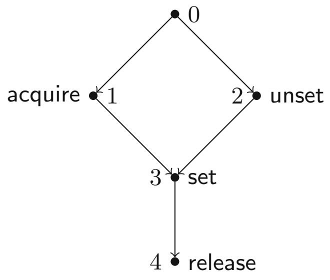

Fig. 1. An example event structure

Executions are modeled by an event structure $(E, l, \lesssim)$, which is a set of events E and a partial order on events $\lesssim$. Events are labeled by a set of actions. The function $l : E \to 2^{\mathcal{A}}$ gives the set of labels for each event. If $E \subseteq E'$ for some event structure $(E', l, \lesssim)$, we write l_E and $\lesssim_E$ for l and $\lesssim$ restricted to the set E, such that $(E, l_E, \lesssim_E)$ is again an event structure. The idea of using an event structure for reasoning about concurrent programs was introduced by Vaughan Pratt in 1986 [12], and we incorporate it for use in a Hoare-style setting.

Figure 1 shows a possible event structure. The nodes indicate events, which are numbered so we can talk about them later. An arrow from node e_1 to e_2 indicates $e_1 \lesssim e_2$, and the set of labels $l(e)$ is written next to each node, omitting the $\{\}$ curly brackets. Arrows that follow from transitivity of $\lesssim$ are not drawn. The intuition behind an event structure is that $\lesssim$ represents the order in which events, and therefore actions, occur.

Construction. We construct event structures in one of three ways: Initialisation, extending an existing structure by a single subsequent action, and by combining parallel events. None of these operations removes anything from event structures, so they grow monotonically, and only through subsequent events. In other words: if e takes the value of an event structure $(E', l', \lesssim')$, then at any later point e holds a value $(E, l, \lesssim)$ such that $E' \subseteq E$, $l_{E'} = l'$, $\lesssim_{E'} = \lesssim'$, and for an event $i \in E, i' \in E'$ such that $i \notin E'$, we have $i' \not\gtrsim i$. This monotonicity is important for reasoning about event structure variables.

Initialisation. Initialisation happens through declaring a variable as an event structure. The variable initializes to an event structure where the set of events is the empty set (this uniquely defines l and $\lesssim$ too). We use the following syntax for this: `var e = new EventStructure();`.

Extension. If $S = \{s_1, s_2, \ldots, s_n\} \subseteq \mathcal{A}$ is a set of actions and `e` is a ghost variable that holds the event structure $(E, l, \lesssim)$, then `e` can be extended by an event with labels S. Let t be a fresh event. We can think of t as a unique timestamp, or as a counter that increases every time we use it. We ensure that t is larger than any of the events in the structure to which we add it. Fresh means that for any two sets of events E_1 and E_2 appearing in our program, any common events must have been created at the same point in our program. In particular, freshness implies $t \notin E$. We define $l' : E \cup \{t\} \rightarrow 2^{\mathcal{A}}$ by $l'(t) = S$ and $l'(i) = l(i)$ for $i \in E$. We define $\lesssim'$ by $t \lesssim' i$ for $i \in E \cup \{t\}$, and $i \lesssim' j \Leftrightarrow i \lesssim j$ for $i, j \in E$. Then the new value of e after extending it with an event with labels S is $(E \cup \{t\}, l', \lesssim')$. We use `v.add(`$s_1, s_2, \ldots, s_n$`);` as syntax for this.

Parallel events. To hand off an event structure to a forked thread, it is allowed to make a copy of a ghost variable indicating an event structure `v`. We write `var w = v.copy();` for this. Any subsequent adding of events to `v` or `w` happens in isolation from each other as described above.

Parallelism becomes visible in the thread structure when threads are joined again, and we will use `union` to join the corresponding event structures. We argue that the ordinary set union suffices: Let $(E_1, l_1, \lesssim_1)$ and $(E_2, l_2, \lesssim_2)$ be events structures. Note that since we only added fresh events, we can define $l : E_1 \cup E_2 \rightarrow 2^{\mathcal{A}}$ by $l(i) = l_1(i)$ for $i \in E_1$ and $l(i) = l_2(i)$ for $i \in E_2$, as any element $i \in E_1 \cap E_2$ must have been created with the same labels: $l_1(i) = l_2(i)$. We write $l_1 \cup l_2$ for l defined this way. Similarly, $\lesssim_1 \cup \lesssim_2$ is again a poset by similar reasoning about freshness. Consequently $(E_1 \cup E_2, l_1 \cup l_2, \lesssim_1 \cup \lesssim_2)$ is again an event structure. We write `v.union(w);` to add the structure of `w` to `v`, after which `v` holds the value as described above.

An Example Program. We show how to combine the constructions mentioned, to create event structures through ghost code. The code below creates the event structure of Fig. 1 as the final structure for `v`.

```
var v = new EventStructure();
v.add();
var w = v.copy();
w.add(acquire); v.add(unset);
v.union(w);
v.add(set); v.add(release);
```

Note that despite the suggestion of parallelism in the `acquire` and the `unset` action, we did not actually use a parallel program to do so. However, changing the execution order of `w.add(acquire);` and `v.add(unset);` would create a similar event structure (equal up to isomorphism).

3 Relation to Processes

Our goal of using a ghost event structure variable is to constrain it by using a class invariant. We introduce processes to constrain the event structures, as a process describes the development of an event structure in an intuitive way.

For ghost variable `v` and a process P, the invariant `inPrefix(v,`P`);` will indicate that at any time, the event structure e that is the value of `v`, $e \in \mathit{prefix}(P)$ holds. To explain what is meant by $\mathit{prefix}(P)$, we introduce the language in which to write P, in Sect. 3.1. We relate event structures to processes by defining what it means for an event structure to be valid for a process, and define the function *prefix*, in Sect. 3.2.

3.1 Processes

A process P is defined using process variables, actions, the empty process, sequential and parallel composition, and nondeterministic choice. Process variables are written $A, B, \ldots \in \mathcal{P}$. We write $\mathsf{a}, \mathsf{acquire}, \ldots \in \mathcal{A}$ to denote actions. We write $P, Q, \ldots$ for processes. A process variable A is defined by stating a declaration of the shape $A = P$, where P is an expression of the shape:

$$P ::= A \mid \mathsf{a} \mid \epsilon \mid P{\cdot}Q \mid P \parallel Q \mid P + Q$$

We require all process variables to be declared exactly once[1]. The precedence of the operations is $\cdot$ over $\parallel$ over $+$, so $((P{\cdot}Q) \parallel R) + S$ does not need any parenthesis.

Here is an example of two process declarations:

$$B = (\mathsf{set} + \mathsf{unset}) \parallel B + \epsilon;$$
$$C = \mathsf{acquire}{\cdot}B{\cdot}\mathsf{release};$$

Process B models any number of arbitrarily ordered setting and unsetting actions. Process C models a process in which such an arbitrary set of actions happens between an acquire and a release.

3.2 Valid and Prefix Event Structures

We define validity to be able to relate event structures to processes. The definition will also be used to define a prefix. We inductively define what it means for an event structure to be a valid structure for a process, given a context of process variable declarations:

- If $(E, l, \lesssim)$ is a valid event structure for the process P, and the process variable $A \in \mathcal{P}$ is declared as $A = P$, then $(E, l, \lesssim)$ is valid for A.
- Let $(E, l, \lesssim)$ be an event structure with exactly one event: $\{e'\} = E$, and $l(e') = \mathsf{a}$. Then $(E, l, \lesssim)$ is valid for a.
- An event structure $(E, l, \lesssim)$ for which $\forall e \in E. l(e) = \{\}$, is valid for ϵ.
- If $(E, l, \lesssim)$ is an event structure, $E_1 \cup E_2 = E$ with E_1 and E_2 disjoint, $(E_1, l_{E_1}, \lesssim_{E_1})$ is valid for P and $(E_2, l_{E_2}, \lesssim_{E_2})$ is valid for Q, then $(E, l, \lesssim)$ is valid for $P \parallel Q$. If additionally $\forall e_1 \in E_1, e_2 \in E_2.\ e_1 \lesssim e_2$, then $(E, l, \lesssim)$ is valid for $P{\cdot}Q$.
- If $(E, l, \lesssim)$ is an event structure that is valid for P, then $(E, l, \lesssim)$ is valid for $P + Q$, as well as for $Q + P$.
- Nothing else is a valid event structure for a process.

[1] Because how validity is defined in the next section, a process defined as $A = A$ is equivalent to the process for which no event structures are valid, not even the empty one.

We could extend the language for processes (say with hiding operations), as well as the event structure (say with a conflict relation), as long as the model checker we use to reason about validity of event structures supports the added constructions.

For l' and $\lesssim'$ such that $(\{0,1,2,3,4\}, l', \lesssim')$ is the event structure indicated in Fig. 1, we get: The event structure $(\{3\}, l'_{\{3\}}, \lesssim'_{\{3\}})$ is a valid event structure for B. Consequently, the event structure $(\{0,1,3,4\}, l'_{\{0,1,3,4\}}, \lesssim'_{\{0,1,3,4\}})$ is a valid event structure for C. However, $(\{0,1,2,3,4\}, l', \lesssim')$ is not a valid event structure for C. It is, however, a valid event structure for $B \parallel C$.

When reasoning about programs, we describe partial executions, which we also relate to processes. A prefix encompasses this idea. A prefix is an event structure that could be extended to become a valid event structure for a process P: Let $(E, l, \lesssim)$ be a valid event structure for P. Take $E' \subseteq E$ such that it is upward closed with respect to $\lesssim$, that is: if $e' \in E'$, $e \in E$ and $e \lesssim e'$, then $e \in E'$. Then $(E', l_{E'}, \lesssim_{E'})$ is a prefix event structure for P. The set of all such prefixes is written $\mathit{prefix}(P)$. Similarly, the set of all valid event structures for P is written $\mathit{valid}(P)$.

3.3 Using Invariants

We use an invariant system to reason about the state of program variables in relation to a ghost variable. The invariants we consider are checked after every change to shared variables: In a valid program, all invariants hold before and after every atomic action. This fine-grained level of invariants allows us to relate processes to a program state. We illustrate this with an example of a lock.

In a program with a spinlock, a single boolean la indicates the availability of the lock. If the lock is available, a thread may atomically compare and swap la from true to false. That thread is then responsible for eventually releasing the lock by setting it back to true. We can model the lock with a very simple process:

$$L = \epsilon + \mathsf{lock}\cdot\mathsf{unlock}\cdot L$$

We use a global variable p to keep track of our locking process. Code for obtaining the lock could look like this (replacing lines 4 to 7):

```
boolean success = false;
while (!success){
  success
   = ( compare_and_swap( la,true,false )
        /*@ atomically {
              if (\result) {e.add(lock);}
            } *@/
      );
}
```

Here the atomic compare and swap operation is executed as a single atomic action together with our ghost code. The block starting with `/*@` and ending with `@*/` is ghost code, and is to be ignored by a compiler, but is 'virtually' executed in symbolic analysis of the code. We put extra brackets around this single atomic action for clarity. This means no other threads can interleave between the compare and swap and the ghost code on line 6. The `\result` on line 6 refers to the return value from the compare and swap operation. Note that we cannot use `success` yet, as the write to `success` does not happen until after this atomic action. A verifier checks that the expression preceding line 6 is indeed atomic, and the code in line 6 is valid ghost code in that it does not change any non-ghost parts.

Now we wish to verify that this code actually maintains $\mathsf{e} \in \mathit{prefix}(L)$. That is: the event structure in the variable `e` is a valid prefix of the process described by L. For this code, that means we need to show that when `e.add(lock);` is virtually executed, the value of `e` is such that adding an event with the label lock preserves the invariant. The invariant guarantees that `e` is in the prefix of L, but that does not suffice to prove what we need to show: The event structure $(\{0, 1\}, \lambda x.\mathsf{lock}, \leq)$ where $\leq$ is the standard order on natural numbers is not in the prefix of L, but can be reached after `e.add(lock);` if the original value of `e` was $(\{0\}, \lambda x.\mathsf{lock}, \leq)$, which is in the prefix of L. This situation should not occur, because of how `la` relates to `e`, but we have not made this explicit yet. We do so in another invariant.

The invariant $\mathsf{e} \in \mathit{valid}(L) \Leftrightarrow \mathsf{la}$ describes that `la` is true if and only if the value of `e` is an event structure in L. As we do the atomic compare and swap, we can prove that both invariants are maintained by case analysis: If the compare and swap fails, `la` is unchanged and so is `e`. Since the invariants holds before the compare and swap, it also does after it. For the other case, the compare and swap succeeds. This means that before the atomic action, `la` was true. Therefore, we must have been in an accepting state of L per our second invariant. We are allowed to do the lock action from that state, which establishes $\mathsf{e} \in \mathit{prefix}(L)$. Additionally, we will end up in a non-valid state of L by doing this action. As `la` is false after the atomic action, we also established $\mathsf{e} \in \mathit{valid}(L) \Leftrightarrow \mathsf{la}$. This shows that the two stated invariants are preserved. The reasoning required to establish this, can be stated as an isolated model checking problem.

Note that our reuse of L in the invariant $\mathsf{e} \in \mathit{valid}(L) \Leftrightarrow \mathsf{la}$ is a bit of a lucky coincidence. The processes $\mathsf{lock}{\cdot}\mathsf{unlock}{\cdot}L$ and $\mathsf{lock} + \mathsf{lock}{\cdot}\mathsf{unlock}{\cdot}L$ all have the same prefixes as L, so we could have used them in the first invariant. However, they differ in $\mathit{valid}(L)$, so they would not be suitable for the invariant that fixes `la`. In certain cases, one would need to write a separate process for different invariants.

3.4 Limitations and Extensions

We illustrate a limitation of our approach by the same example of a lock. This time, we focus on the release of the lock, rather than the acquire. We could use the same solution as for the lock, but there is a subtlety: While an acquire requires a compare and swap operation, a release can be done with the unconditional assignment la=true;. Our approach can be extended in several ways, which we will sketch now. We end this section by briefly discussing which option would be the best choice to implement in an existing tool.

The invariant we need to prove the unconditional release preserves the invariant is as follows: Only the thread or process that acquired the lock is allowed to release it. We can state this invariant in terms of permissions: every thread can do a lock action, after which it obtains permission to do an unlock action. Another way to state this invariant is in terms of a rely-guarantee invariant: all threads must guarantee to do a lock before any unlock. Finally, we could change the definition of our process to a thread-oriented version, making explicit which thread does the lock in the process.

Using Permissions. For using permissions, we assign permissions to actions. This ties in nicely to verification tools that already use permissions. The idea is to introduce a new permission (or resource), which we call can_unlock. In an implementation, the permission itself can be left undefined, or the write permission to an arbitrary heap location can stand in its place. We will give this permission to the thread that can perform an unlock action, which means we will need to prove that at most one thread can get that permission.

In this solution, add pre- and postconditions to e.add(lock) and e.add(unlock): As a postcondition for e, you gain the permission can_unlock. The permission can_unlock is a precondition to adding the unlock event to e. Aside from the invariants, adding a lock event to e has no preconditions, and adding the unlock has no postconditions. It follows from e $\in$ *prefix*(L) (by model checking L) that the number of outstanding can_unlock permissions is at most one. Crucially, this means that at most one thread has the can_unlock permission. This should allow us to prove that no unlock events are added to e as long as we hold can_unlock.

Using rely-guarantee. Using a rely-guarantee mechanism, we state that every thread, and therefore also the environment of a single thread, must do a lock before an unlock. Together with the invariant e $\in$ *prefix*(L), this means that the environment of some thread cannot do an unlock after our thread performed a lock. For this approach to work one needs to tie the execution of threads to that of method calls: When a thread is forked, it gets assigned a process that acts as its contract. Assigning a process to a forked thread as a method is worked out under the name 'abstract models' as currently implemented in the tool Vercors [11]. A similar principle might be usable to also state properties about the environment of a thread. Indeed, the combination of using separation logic and rely/guarantee based reasoning has been proposed by Vafeiadis et al. [14].

Thread-Specific Event Structures. Finally, one could add a ghost event structure variable $t(n)$ for each thread n. For each of these variables, we have $t(n) \in \mathsf{prefix}(L)$. Each thread n gets exclusive access to $t(n)$. Additionally, we add an invariant that states that e is an interleaving of all $t(n)$. This solution works the same way as the solution of using permissions, with the difference that having write access to a $t(n)$ that contains a lock event here takes the role of having the can_unlock permission. Again, the invariants collectively guarantee that only one thread at a time has this permission.

This solution seems to be really close to what we described in this paper so far. The main addition is to be able to express the composition of a set of thread-specific variables $t(n)$. We can use the union command, which does this for two variables. However, we need to compose an unbounded number of variables, rather than two as with union. This seemingly small detail hinders the use of model checkers at the place where we intend to use them.

Future Directions. In this section, we described three ways to verify correctness of the unlock. Each has its own benefits: there is a clear path for the implementation of the first solution, the second solution seems to best match existing literature, and the third solution seems to constitute the smallest change in the language of existing tools. We do not know whether the most convenient solution is among these three, or which of these would be the best for a tool user. We hope to discuss these directions with the participants of ISoLA 2018.

4 Conclusions and Future Work

We described the use of a new kind of ghost variable to help verify programs in an intuitive way. This gives us a way to reason about programs as if they were state machines, in a way that allows us to choose the abstraction level ourselves. Invariants allow us to tie programs into program variables, such that the reasoning also helps us to state properties about the program based on that reasoning.

We believe the ideas in this paper can be implemented by combining model checking and existing static verification tools, but have not yet worked out all necessary details on how to do so. Details on how to do this in Vercors remain future work. We hope this paper inspires readers to come up with different ways of implementing these ideas in other tools as well.

A future direction of research is to determine whether we can use this approach to verify properties that can only be stated on a process level, like the linearizability of methods. Linearizability is an important property for high performance libraries. To prove it, one would associate each method of a certain class with a single action. We would like to be able to assert that regardless of how we call these methods in instance of that class, the event structure variable registers each of those actions exactly once, sequentially. Furthermore, the sequential execution of the corresponding methods should give the same state for the instance. A proof of linearizability of a set of methods allows us to treat those methods as atomic actions themselves, giving an extra opportunity of making proofs more modular.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
2. Amighi, A.: Specification and verification of synchronisation classes in Java: a practical approach. Ph.D. thesis, University of Twente (2018)
3. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
4. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_15
5. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: component reasoning for concurrent objects. J. Logic Algebraic Program. **81**(3), 227–256 (2012). https://doi.org/10.1016/j.jlap.2012.01.003, The 22nd Nordic Workshop on Programming Theory (NWPT 2010)
6. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 315–331. Springer, Heidelberg (2008)
7. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585
8. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. CoRR abs/1405.4028 (2014). http://arxiv.org/abs/1405.4028
9. Lamport, L., Owicki, S.: Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst. **4**(3), 455–495 (1982). https://www.microsoft.com/en-us/research/publication/proving-liveness-properties-concurrent-programs/
10. Oortwijn, W., Blom, S., Huisman, M.: Future-based static analysis of message passing programs. In: PLACES, pp. 65–72 (2016)
11. Oortwijn, W., Blom, S., Gurov, D., Huisman, M., Zaharieva-Stojanovski, M.: An abstraction technique for describing concurrent program behaviour. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 191–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_12
12. Pratt, V.: Modeling concurrency with partial orders. Int. J. Parallel Program. **15**(1), 33–71 (1986). https://doi.org/10.1007/BF01379149
13. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inf. **63**(4), 385–410 (2004). http://dl.acm.org/citation.cfm?id=2370686.2370691
14. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
15. Vafeiadis, V.: Formal reasoning about the C11 weak memory model. In: Proceedings of the 2015 Conference on Certified Programs and Proofs, pp. 1–2. ACM (2015)